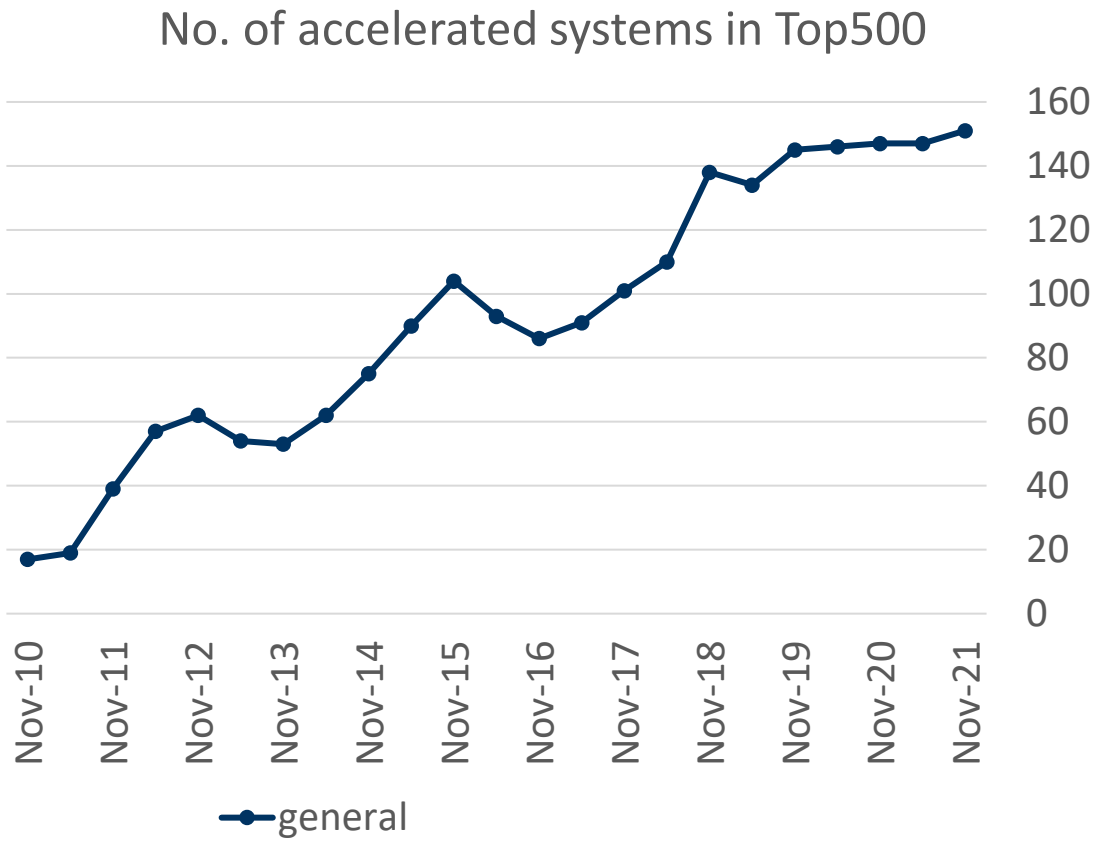
703308 VO High-Performance Computing
Accelerator Computing

Philipp Gschwandtner

Motivation

- Accelerator market share in HPC has been steadily increasing and will likely continue to do so
  - 10 out of top 10 on Green500 list (Nov. 2022)
- distributed memory clusters with accelerators provide some of the best cost- and energy-efficiency in HPC
- Application developers **need** to use accelerators to get high performance on modern systems



Motivation

- More difficult than “regular” parallelism
  - different memory address spaces (up to now)
  - completely different hardware architecture

- “Understanding hardware is important to understand performance.” (Lukas Einkemmer)
  - even more significant in GPU and heterogeneous-hardware computing

Overview

- Hardware
  - very brief crash course
- Software
  - mainly focusing on SYCL
- Current hardware trends

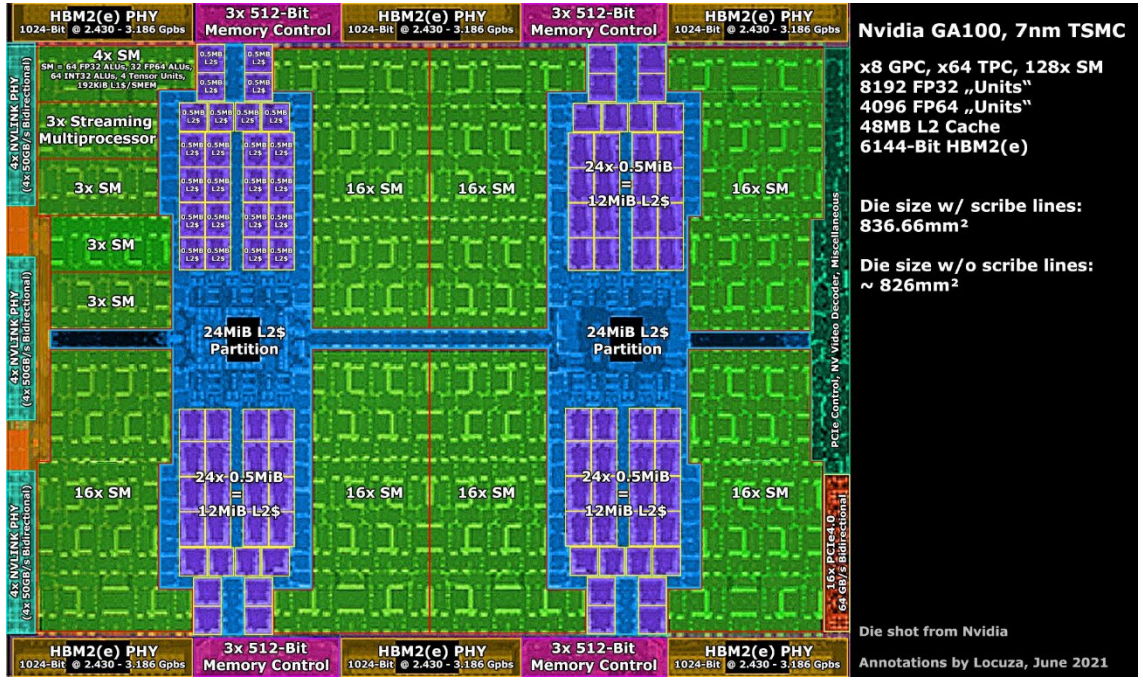
Accelerator technologies

- multiple accelerator technologies for different use cases
  - GPUs: originally graphics processing, recently (~15 years) scientific computing
  - TPUs: deep learning & linear algebra
  - DPUs: big data processing
  - FPGA: programmable hardware for varying workloads
  - etc.
- We'll mostly focus on GPUs for this lecture!
- specific properties common to all accelerators
  - dedicated chips with different ISA
  - dedicated memory (e.g. GDDR6, HBM)
  - connected via some link to the "host" CPU (e.g. PCIe)

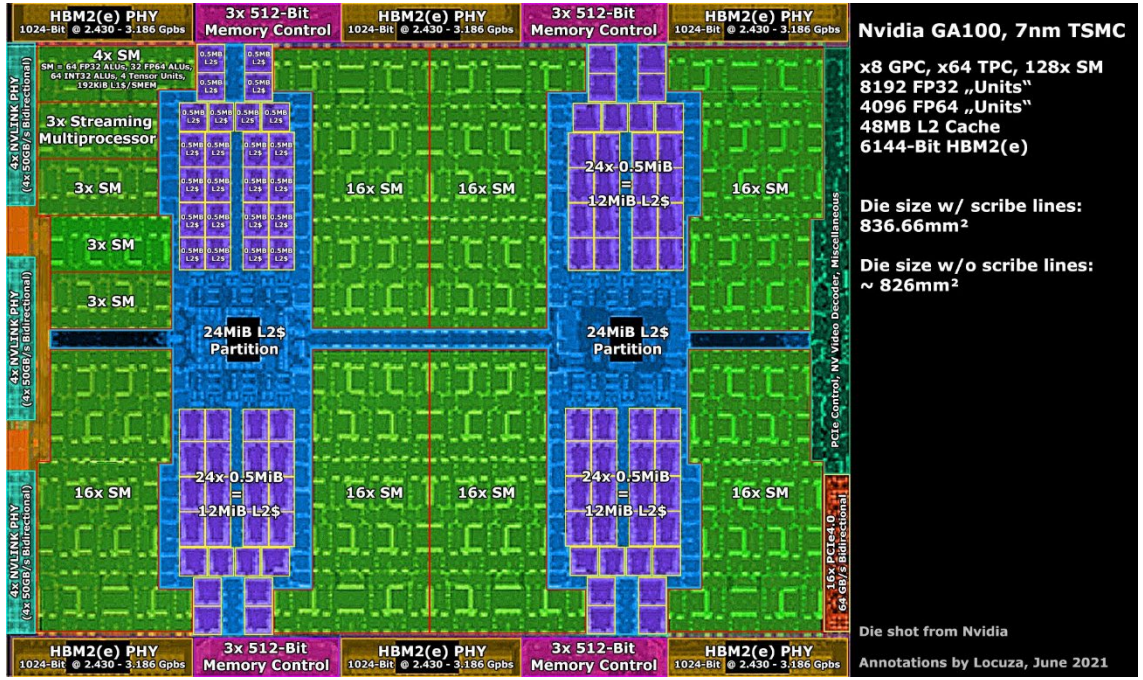
GPUs

- high-performance computing hardware…
  - high instruction throughput
  - high memory bandwidth
- …for massively parallel problems only
- GPUs and CPUs are different points on the same hardware design tradeoff
  - CPU: few threads with high per-thread performance
  - GPUs: thousands of threads with low per-thread performance

Nvidia GA100 GPU die compared to Intel Alder Lake CPU

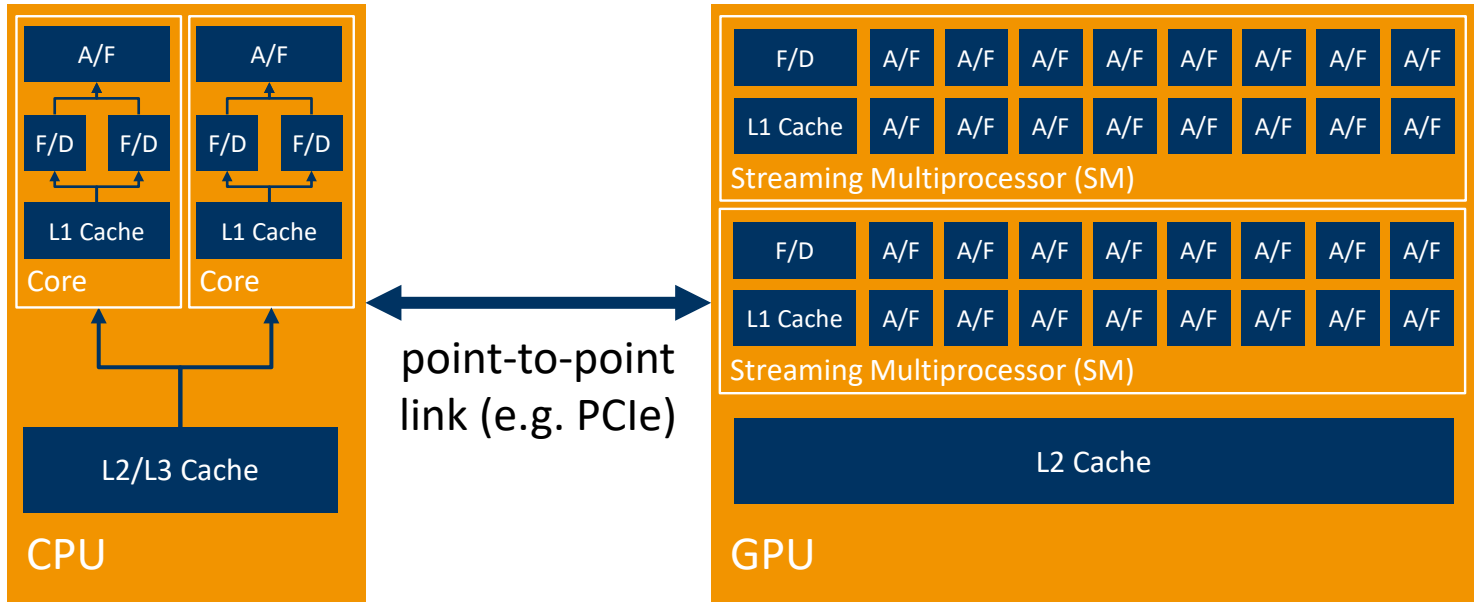


Relative die size comparison (approximated)



GPU hardware

- a “core” consists of multiple hardware threads
  - some of which share a program counter, e.g. must execute SIMD-style (newer architectures are more flexible)
- further subdivision of core ➔ block ➔ thread common
  - lots of details…

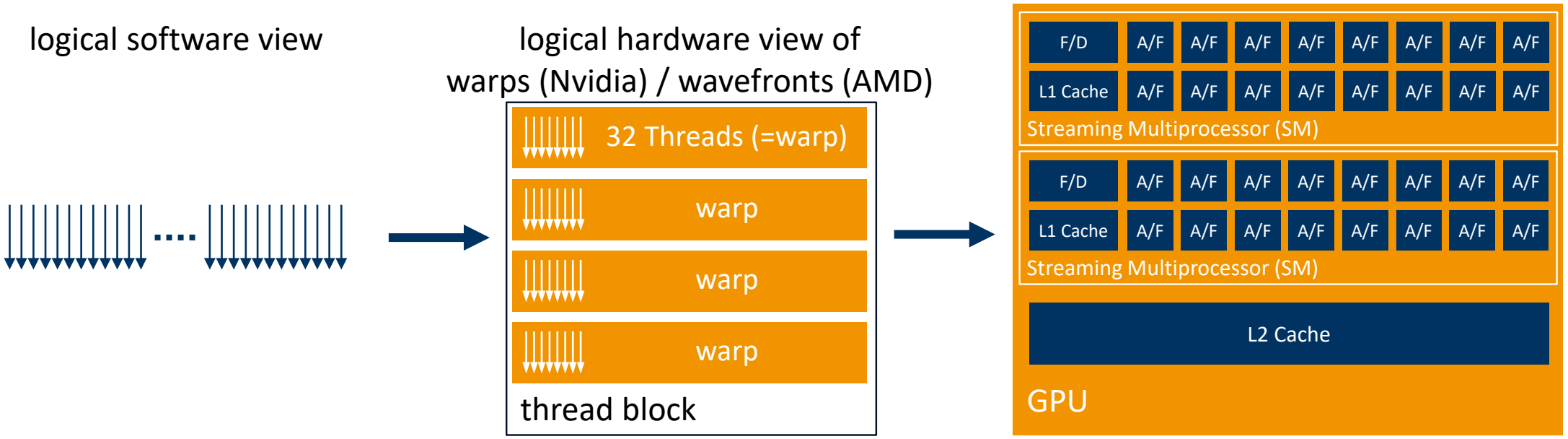


Terminology

- Programming models and hardware vendors use different terminology to largely mean the same thing

CUDA	SYCL	OpenCL
SM	CU	CU
SM core	PE	PE
thread	work item	work item
block	work group	work group

Mapping software to hardware



- hardware always deals in fixed-sized numbers of threads (e.g. 32)
  - You requested 7 threads? hardware runs 32 and discards the result of the last 25
  - You requested 33 threads? hardware runs 64 and discards the result of the last 31

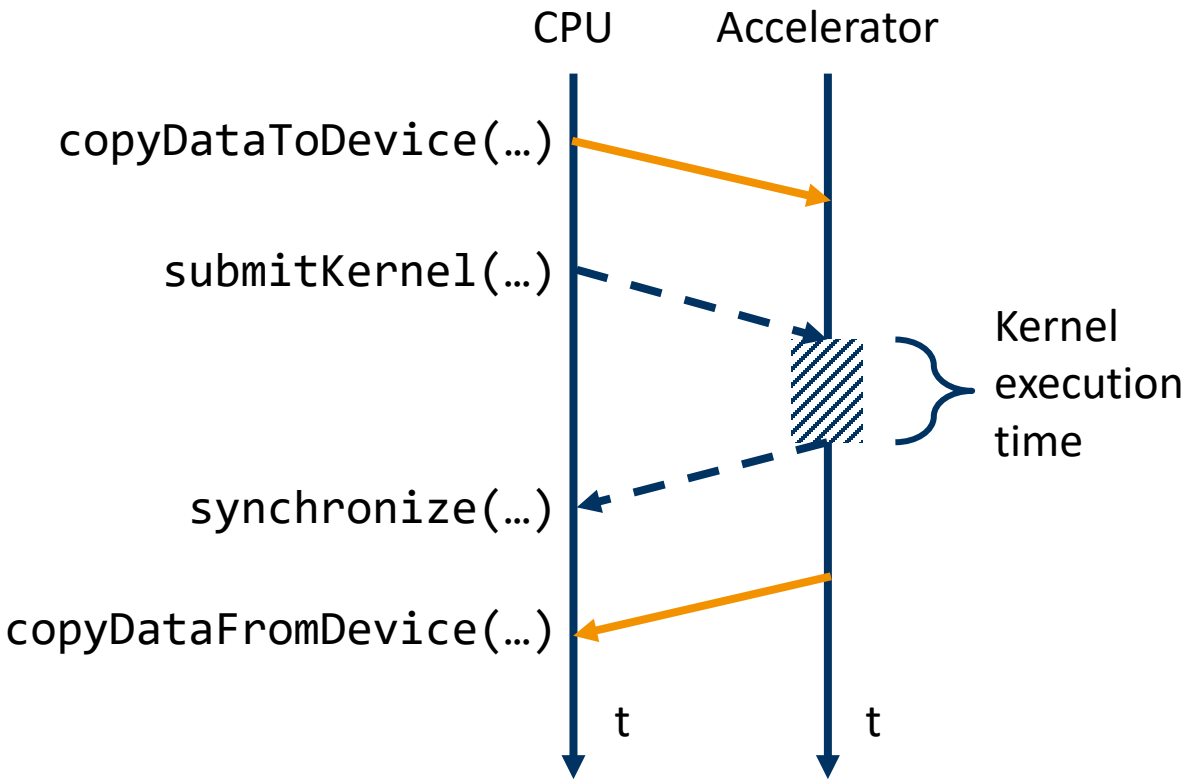
Key hardware concept differences w.r.t. CPUs

- GPUs are very good at frequent context switching for latency hiding
  - run a couple of threads
  - suspend them (e.g. when stalling due to data access)
  - run a couple of other threads
  - do this for a total of several thousand threads
- GPUs are very bad at irregular memory access
  - ideally, try to have coalesced memory access, meaning contiguous work items access contiguous memory addresses
- explicit “caches”
  - called shared memory or local memory (depending on programming model)
  - need to be managed explicitly by the user
- precision can affect performance a lot
  - double-precision vs. single-precision: up to a factor of 64!
  - there AI-specific datatypes (e.g. INT4)

Accelerator Programming Models

Key concept: asynchronous execution

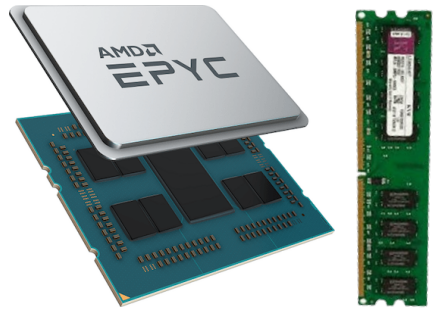
- Accelerators are dedicated processing units
  - have their own clock rate, memory controllers, memory, etc.
  - they execute programs asynchronously
  - host only tells the device when to do what and later checks for the result
- Implications
  - kernel submission time should not be mistaken for kernel execution time



Key concept: different memory address spaces

- Accelerators usually have dedicated memory

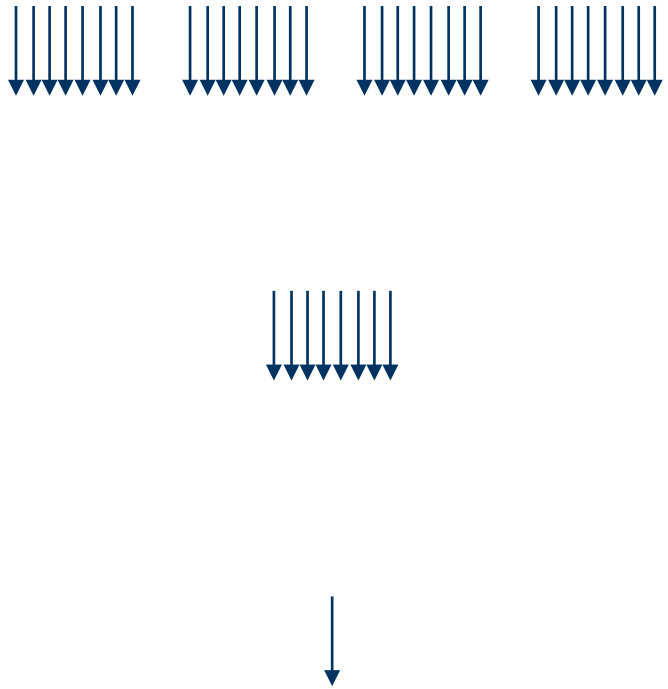
- implications
  - functional: cannot be accessed directly from the host, requires copy operations (sometimes handled automatically, e.g. CUDA unified memory)
  - performance: communication between host and device is expensive, keep at minimum



PCIe 4.0/5.0 / CXL / SXM / NVLink

Programming model

- kernel
  - consists of multiple thread blocks, all working on the same problem in a different iteration space range
  - all thread blocks within a kernel often referred to as “grid”
- thread block
  - logical grouping of threads
  - multiple thread blocks execute independently (can be in parallel, can be in sequence, depending on available hardware resources)
  - are subdivided into multiple warps/wavefronts
- thread
  - single, sequential executing entity



Motivation for using SYCL

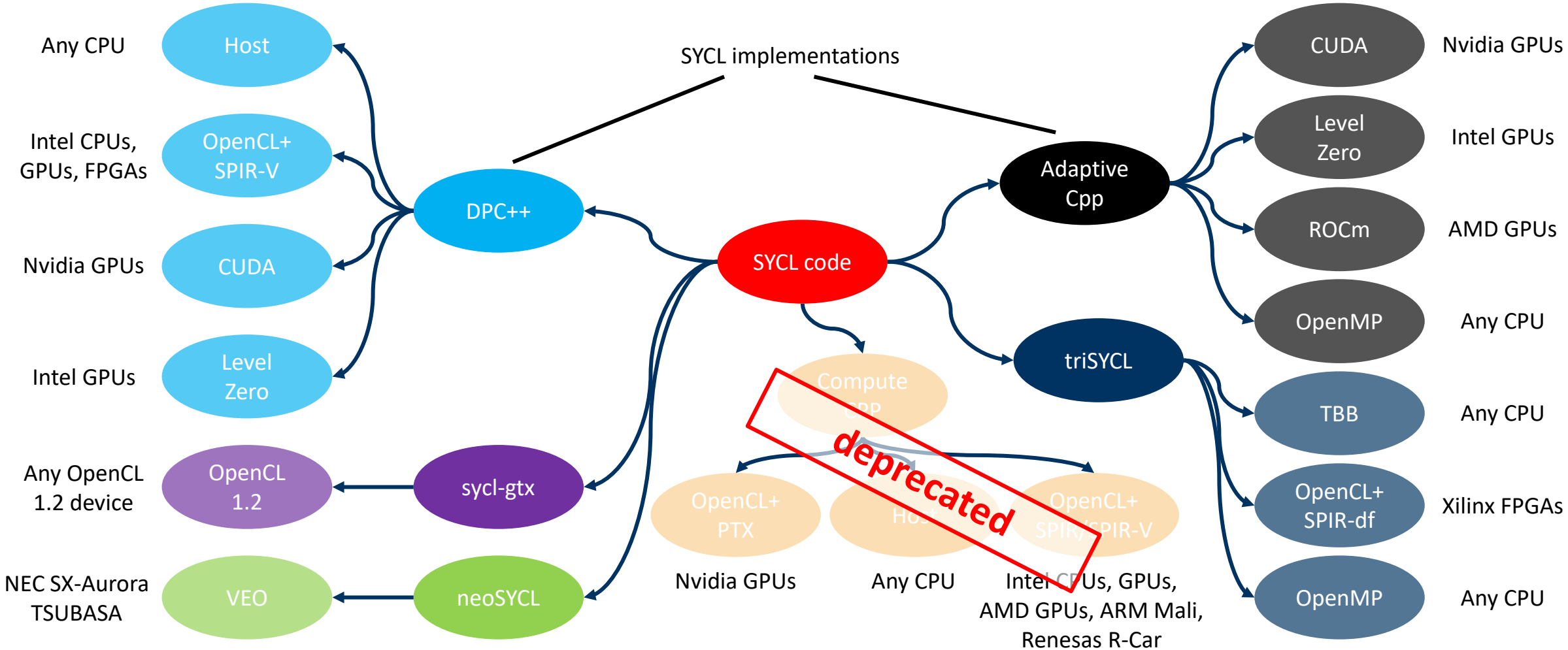
- CUDA: very mature but limited to Nvidia, language extension

- Any alternatives?
  - OpenCL: lots of boilerplate, often deprecated, outdated by today's standards
  - ROCm/HIP: limited to AMD, language extension
  - C++ AMP: deprecated since VS 2022, language extension
  - OpenMP: supports accelerators, requires decent amount of optimization
  - etc…
  - SYCL!

SYCL

- Khronos industry standard
  - successor of OpenCL
- programming model / specification, not an implementation
  - several implementations available, see next slide
  - not limited to a specific vendor or even accelerator type (e.g. FPGAs)
- single-source (!), high-level, C++-based, programming model specifically tailored towards accelerator computing
  - essentially an embedded DSL within C++
- Leverages C++ semantics to provide abstractions for boilerplate code
  - data migration from/to accelerator, kernel specification, etc.

SYCL implementations, software and hardware targets



Code example: vector addition in SYCL

```
#include <CL/sycl.hpp>

cl::sycl::queue q;

cl::sycl::buffer<float, 2> buf_a(host_a.data(), cl::sycl::range<2>(512, 512));
cl::sycl::buffer<float, 2> buf_b(host_b.data(), cl::sycl::range<2>(512, 512));
cl::sycl::buffer<float, 2> buf_c(host_c.data(), cl::sycl::range<2>(512, 512));

q.submit([=](cl::sycl::handler& cgh) {
  auto r_a = buf_a.get_access<cl::sycl::access::mode::read>(cgh);
  auto r_b = buf_b.get_access<cl::sycl::access::mode::read>(cgh);
  auto w_c = buf_c.get_access<cl::sycl::access::mode::write>(cgh);
  cgh.parallel_for<class KernelName>(cl::sycl::range<2>(512, 512),
    [=](cl::sycl::item<2> item) {
      w_c[item] = r_a[item] + r_b[item];
    }
  );
});
```

Basic SYCL concepts

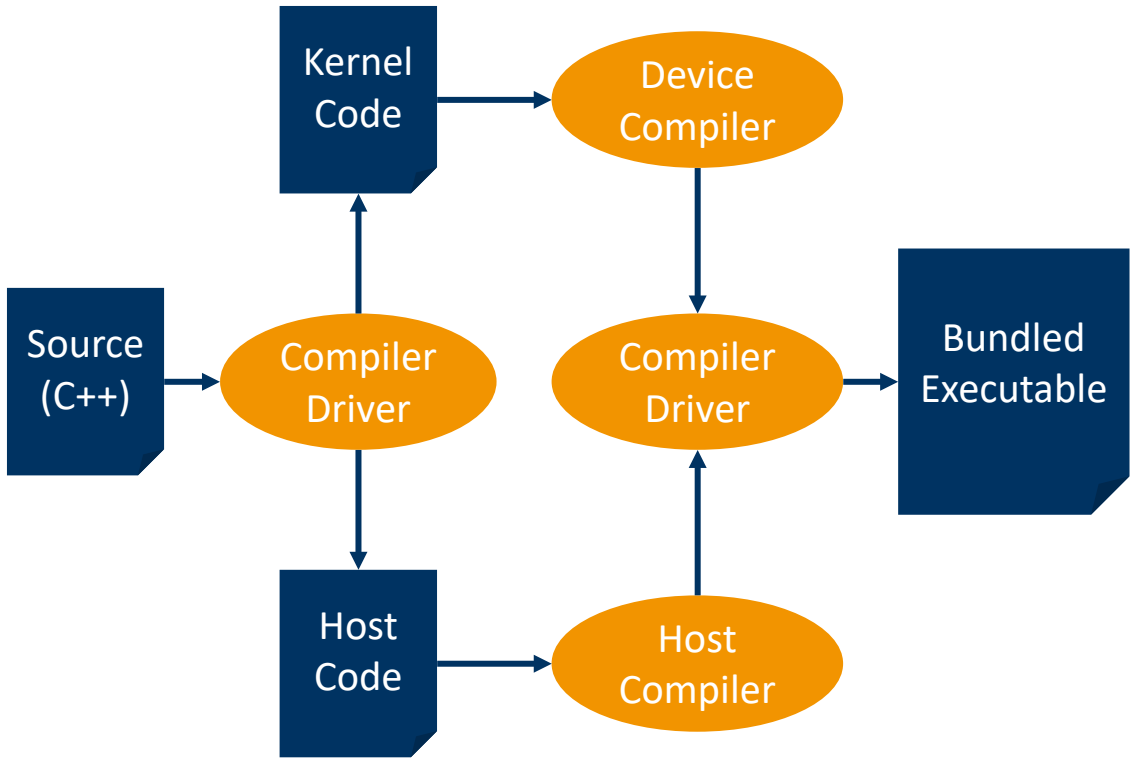
```
queue q;

buffer<float, 2> buf_a(host_a);
buffer<float, 2> buf_b(host_b);
buffer<float, 2> buf_c(host_c);

q.submit([=](handler& cgh) {
  auto r_a = buf_a.get_access<access::mode::read>(cgh);
  auto r_b = buf_b.get_access<access::mode::read>(cgh);
  auto w_c = buf_c.get_access<access::mode::write>(cgh);
  cgh.parallel_for<class KernelName>(range<2>(512, 512),
    [=](item<2> item) {
      w_c[item] = r_a[item] + r_b[item];
    }
  );
});
```

- **queue**
  - controls device activities
  - submission of command groups
  - synchronization with host code
- **buffers**
  - encapsulate 1D – 3D dense, typed data
  - are handles to data in device memory
  - data movements to/from device are handled via explicit **accessors** that specify read/write behaviour, creating an accessor means the underlying data is possibly moved
- **command groups**
  - hold the actual kernel function, its range and accessors
- **kernel**
  - holds the actual body function with work to be done
  - will be executed on the device

SYCL compilation flow (e.g. AdaptiveCpp)

- Compiler driver invokes both host and device compiler respectively
  - can be architecture-specific, e.g. AMD HIP
  - allows compiling for multiple architectures simultaneously into a single executable
  - host compilation pass requires only C++
- Driver then merges compiled device and host binaries into single executable
  - kernels are mapped to the host code at the correct location via their names (C++ type names)

```
parallel_for<class KernelName>
```



Accessors

- Used to access buffers within a kernel
  - buffers are always accessed through an accessor
  - enables compile-time safety checks for e.g. invalid writes of read-only data
  - enables runtime system to establish data flow information
    - used to generate a dependency graph for executing multiple kernels in parallel
    - enables implicit data transfers, e.g. copy-back to host buffers

```
std::vector<float> host_a(size);
{
  buffer<float, 2> buf_a(host_a);
  auto w_a = buf_a.get_access<access::mode::write>(cgh);
  // ... submit a kernel that works with w_a
} // end of object lifetime of buf_a, data is automatically copied back to host_a
```

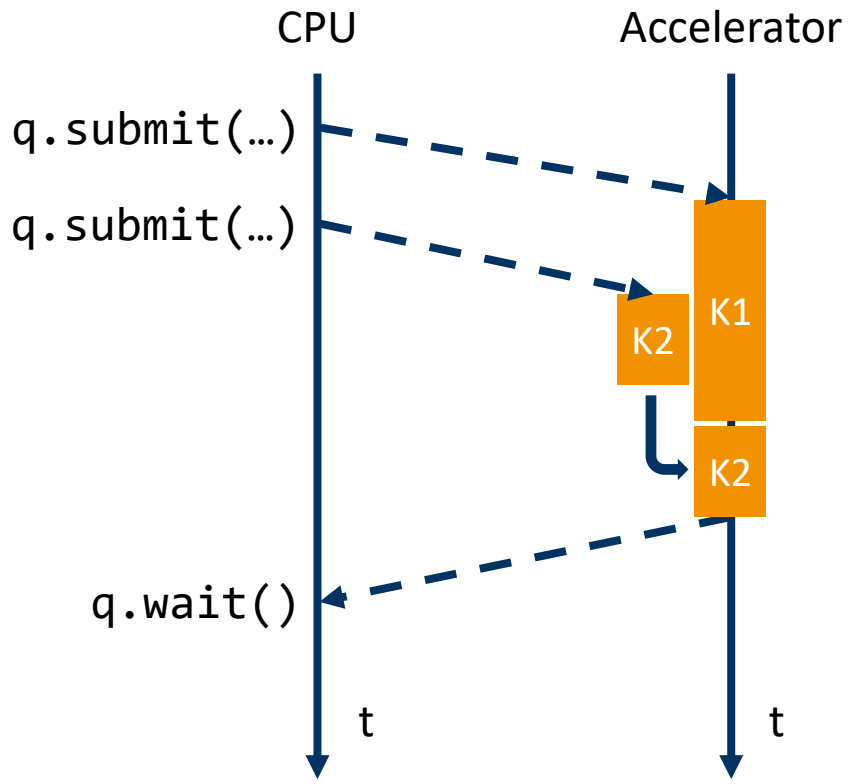
Command group class handlers

- `parallel_for`
  - most often used, provides work group and work item parallelism
  - also: `parallel_for_work_group`
  - also: `parallel_for_work_item`
- single_task
  - sequential execution on a single work item of a single work group
  - rarely used
- copy
  - for copying data from a to b
- fill
  - for initializing data

Consider the API reference guides (=cheat sheet)
https://www.khronos.org/files/sycl/sycl-121-reference-card.pdf
https://www.khronos.org/files/sycl/sycl-2020-reference-guide.pdf

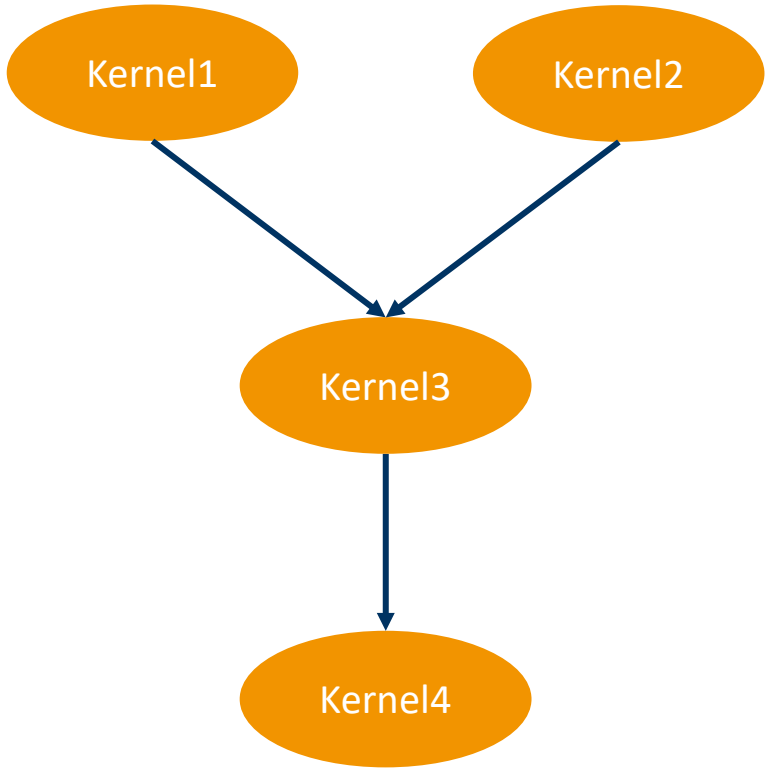
Queue

- controls device activities & synchronizes host and device
- multiple submissions are either scheduled in parallel or in sequence
  - depending on their buffer accessors
    - read/read to same buffer are parallel
    - read/write write/read write/write are in sequence



Queue parallelism & accessor example

```
q.submit([=](handler& cgh) {
  auto r_a = buf_a.get_access<access::mode::read>(cgh);
  auto w_b = buf_b.get_access<access::mode::write>(cgh);
  cgh.parallel_for<class Kernel1>(...);
});
q.submit([=](handler& cgh) {
  auto r_a = buf_a.get_access<access::mode::read>(cgh);
  auto w_c = buf_c.get_access<access::mode::write>(cgh);
  cgh.parallel_for<class Kernel2>(...);
});
q.submit([=](handler& cgh) {
  auto r_b = buf_b.get_access<access::mode::read>(cgh);
  auto r_c = buf_c.get_access<access::mode::read>(cgh);
  auto w_d = buf_d.get_access<access::mode::write>(cgh);
  cgh.parallel_for<class Kernel3>(...);
});
q.submit([=](handler& cgh) {
  auto w_d = buf_d.get_access<access::mode::write>(cgh);
  cgh.parallel_for<class Kernel4>(...);
});
```



Parallelizing loop iteration ranges using work items

```
for(int i = 0; i < 512; i++) {
  for(int j = 0; j < 512; j++) {
    C[i][j] = A[i][j] + B[i][j];
  }
}
```

```
cgh.parallel_for<class KernelName>(range<2>(512, 512),
  [=](item<2> item) {
    int i = item_id.get_access(0);
    int j = item_id.get_access(1);
    C[i][j] = A[i][j] + B[i][j];
    // or even shorter:
    C[item] = A[item] + B[item];
  }
);
```

- loop ranges become work item ranges
  - one point in n-dimensional iteration space becomes a single work item
  - work-items are grouped into work groups and executed in parallel automatically

Further limitations

- recursion
  - generally not available
  - some forms, e.g. tail recursion, can be converted into loops
- exceptions
  - break normal control flow
- directly attaching debuggers
  - greatly depends on the SYCL implementation & backend
  - some vendors support this (e.g. Nvidia CUDA) through specialized tools

- SYCL 2020 does not support offsets anymore

```
for(int i = 1; i < SIZE - 1; i++) {
 // ...
}
```

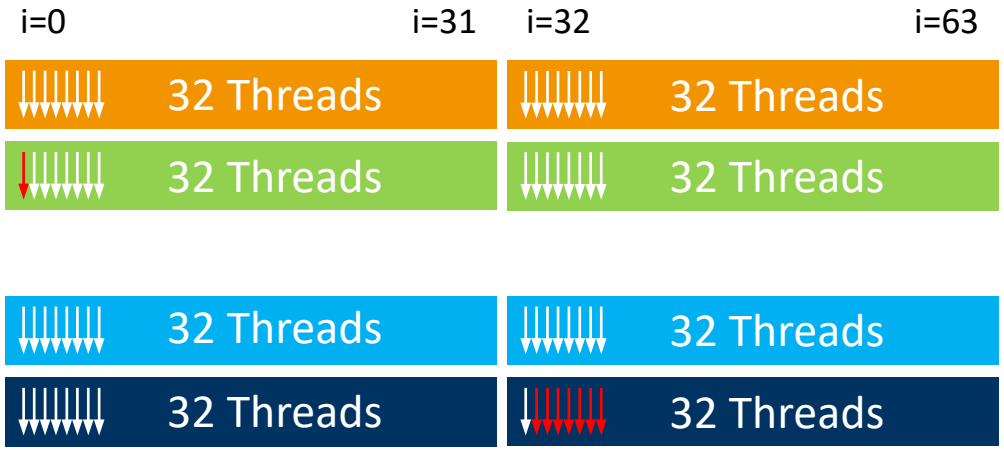
```
cgh.parallel_for<class KernelName>(range<1>(SIZE - 2),
  [=](item<1> item) {
    int i = item_id.get_access(0) + 1;
    // ...
});
```

Caution: Divergent execution

- Many GPUs require all threads within a warp to execute the same instruction stream in complete lock step
  - index-based conditionals forces threads to compute data which is then discarded
  - can pose a performance problem

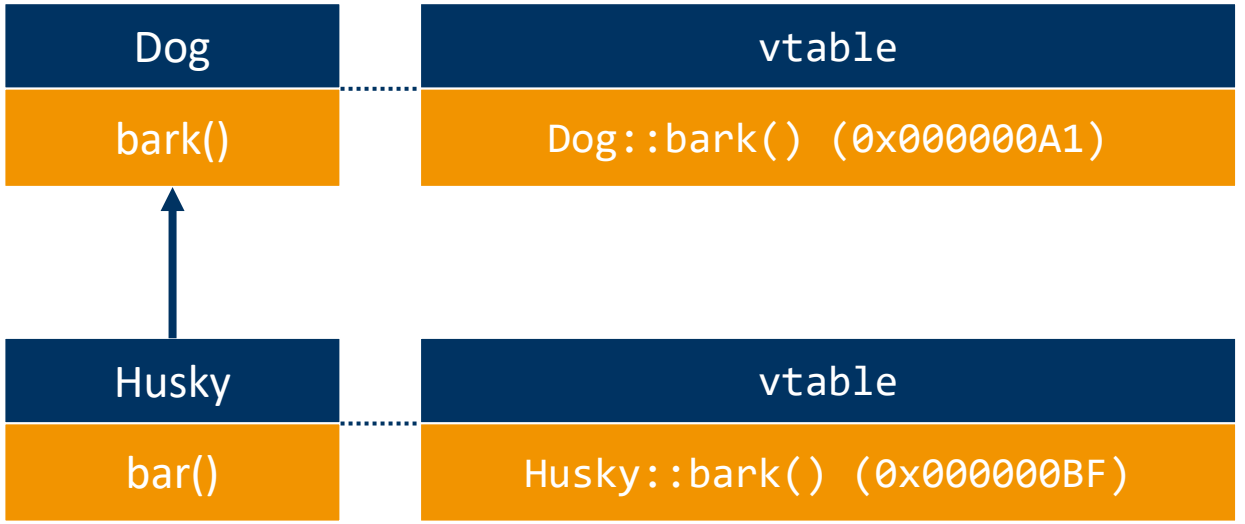
t=0 →
t=1 →
t=2 →
t=3 →

```
if(i > 0) {
 // left boundary condition
}
if (i < size-1) {
 // right boundary condition
}
```



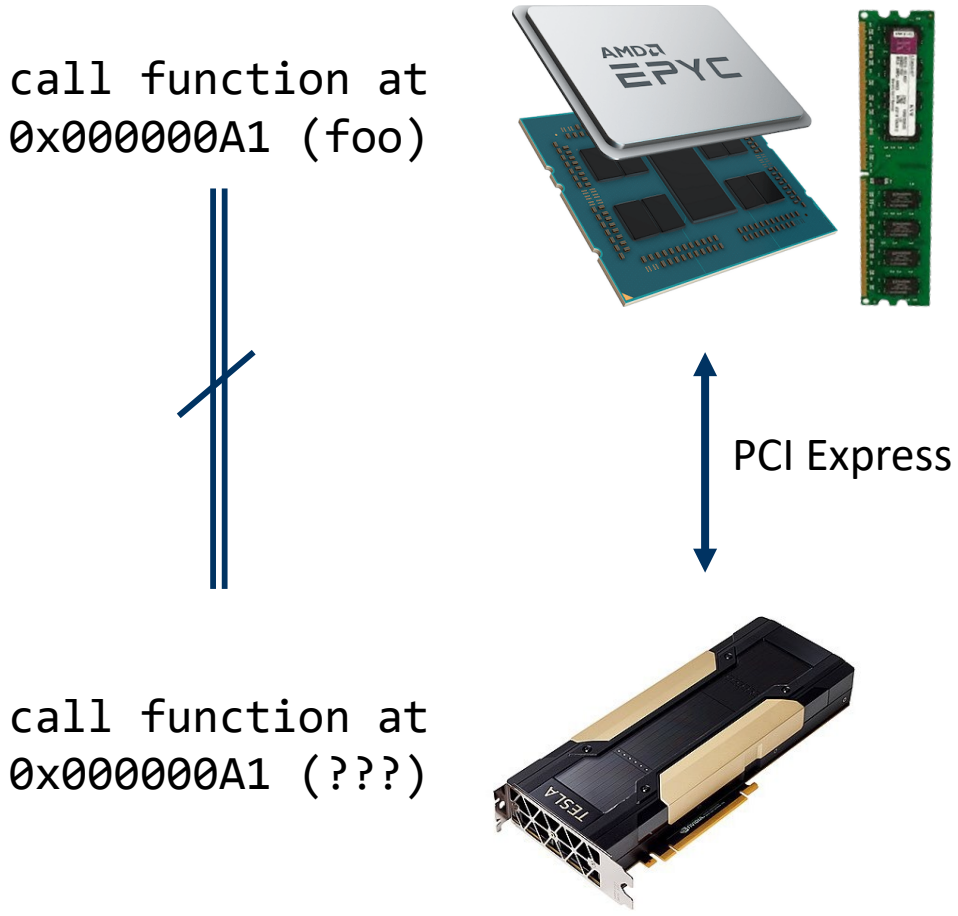
Virtual Function Calls in C++

- Virtual function calls in C++ are realized via pointers
  - every class has a virtual function table (vtable)
  - depending on the runtime-type of your object, a function at a different address might be called



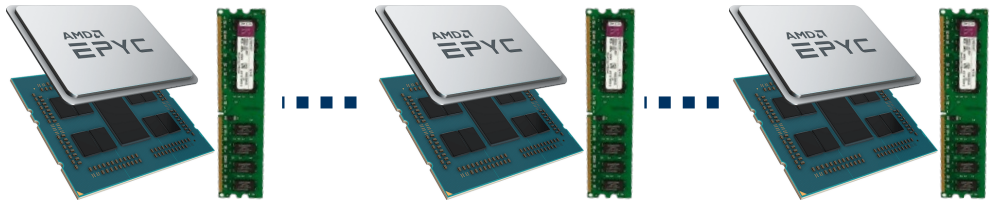
Caution: Virtual Function Calls in SYCL (and CUDA, …)

- CPU RAM and GPU VRAM are distinct memory address spaces
- pointers become meaningless when transferred between RAM and VRAM (same as shallow data copies in MPI)
  - requires some form of mitigation
  - C++ template patterns (CRTP), pseudo-virtual function calls using enumerations, etc…
- Only affects code running on GPUs
  - host-only code can use virtual function calls without issues



Accelerator Cluster Programming

- Combines two very challenging development aspects
  - *distributed memory programming*
  - *accelerator computing*



+

- Current mainstream approaches
  - libraries or skeleton frameworks that abstract computation (Limited to specific domains, often hard to extend)
  - “MPI + X”, with “X” often being CUDA (Requires users to deal with **both** complexities of distributing data and a relatively low-level accelerator API)
- Celerity is positioned in between these extremes

The Celerity idea

- A **high-level API** designed from the ground up for accelerator clusters
  - allows to constrain data structures and processing patterns to ones efficient on accelerators
    ➔ less complex than fully general distributed memory programming
- Based on the SYCL Khronos industry standard
  - single-source, modern C++ for accelerators (“embedded DSL”)
  - designed to run on most hardware supported by OpenCL
  - also supports native CUDA and AMD ROCm execution via hipSYCL

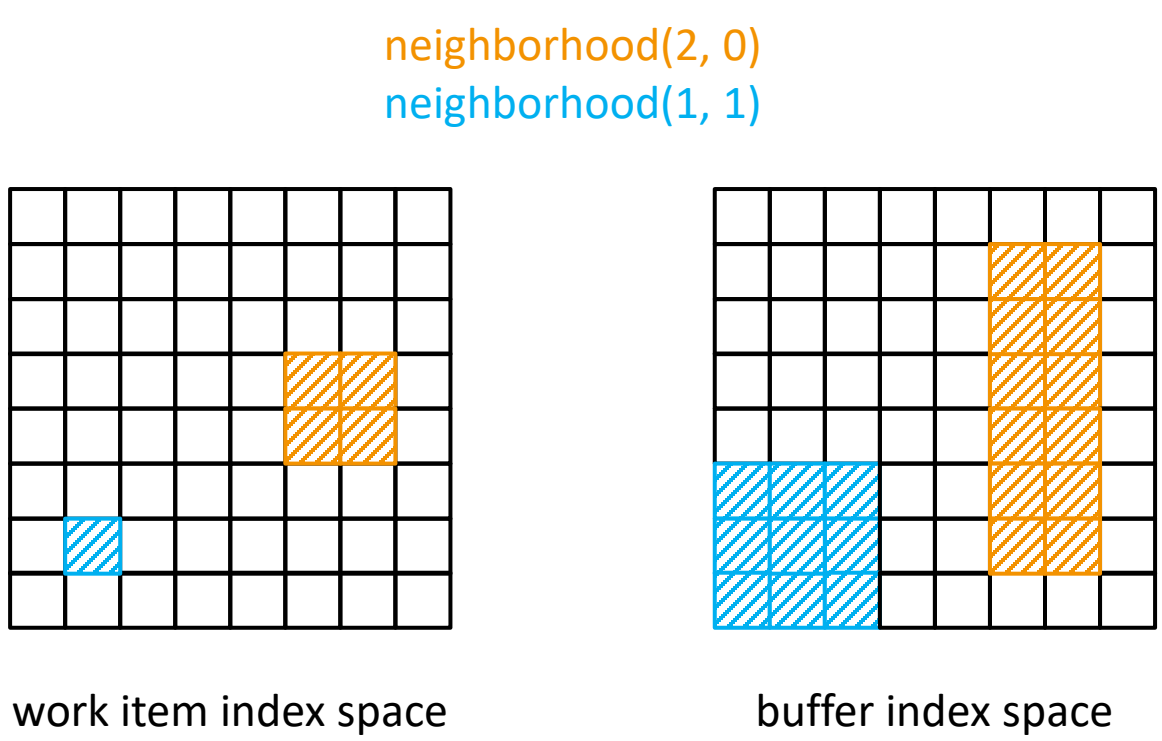
- Attempt to mitigate adoption issues faced by many frameworks
- The goal is **not** to beat the performance of year-long manually tuned applications

Code example: vector addition in Celerity

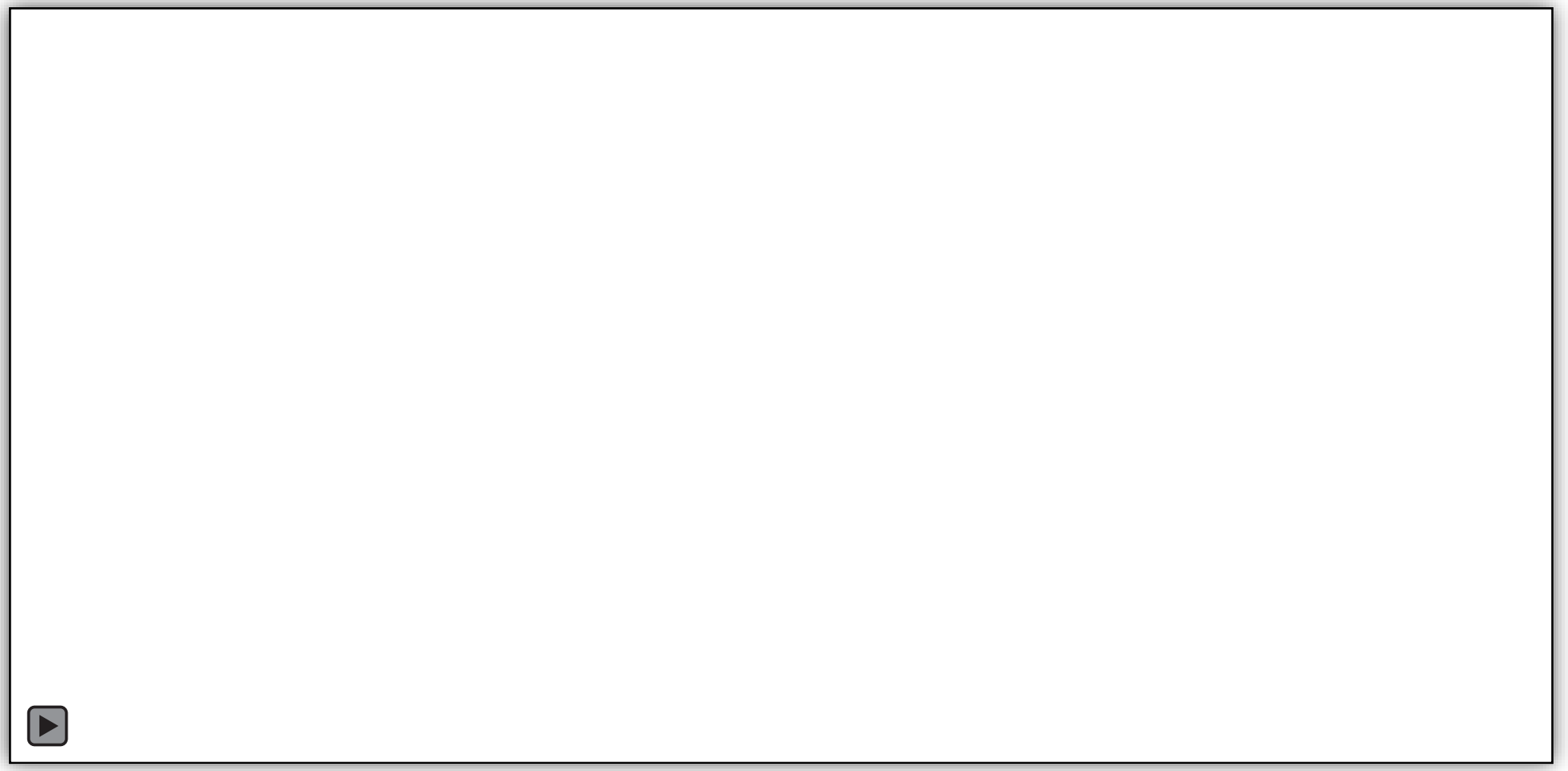
```
celerity::distr_queue queue;

celerity::buffer<float, 2> buf_a(host_a);
celerity::buffer<float, 2> buf_b(host_b);
celerity::buffer<float, 2> buf_c(host_c);

queue.submit([=](celerity::handler& cgh) {
  auto one_to_one = celerity::acccess::one_to_one<2>();
  auto r_a = buf_a.get_access<acc::read>(cgh, one_to_one);
  auto r_b = buf_b.get_access<acc::read>(cgh, one_to_one);
  auto r_c = buf_c.get_access<acc::write>(cgh, one_to_one);
  cgh.parallel_for<class KernelName>(sycl::range<2>(512, 512),
    [=](sycl::item<2> itm) {
      w_c[itm] = r_a[itm] + r_b[itm];
  });
});
```

Celerity range mappers

- arbitrary functors mapping from a K-dimensional kernel execution range chunk to a B-dimensional buffer subrange
  - allows the runtime system to determine data requirements for specific kernel execution range
  - provides work/data location information for distributed memory execution

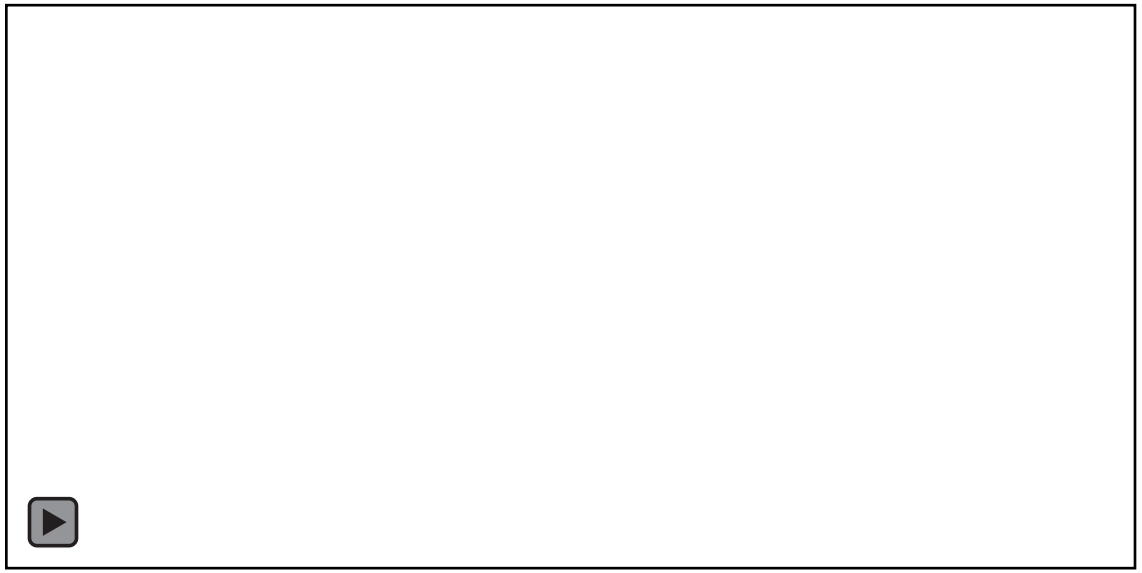


Celerity execution procedure



Cronos (yes, again!)

- Structured grid simulation (finite volume) in astrophysics
  - gamma ray emissions of binary star system LS 5039
- Self-written C++14 code
  - astro- and Particle Physics @ UIBK (Ralf Kissmann, David Huber et al.)
  - MPI-only (no GPUs!)



Density in a binary star system

Porting considerations for celerity and GPUs

✔ structured grid algorithm with minimal global communication

- ✓ excellent scalability on up to 1000 cores (>95% parallel efficiency)
- ✓ used in an ongoing PRACE access project on 12288 cores

✔ Few external dependencies

- ✓ C++-14-compliant compiler, MPI, HDF5, GSL

✘ 40k lines of C++98-style code

- ✘ a ton of manual memory management (`new` & `delete`)
- ✘ a ton of virtual function calls
- ✘ custom vector and matrix classes with runtime-defined index ranges, e.g.

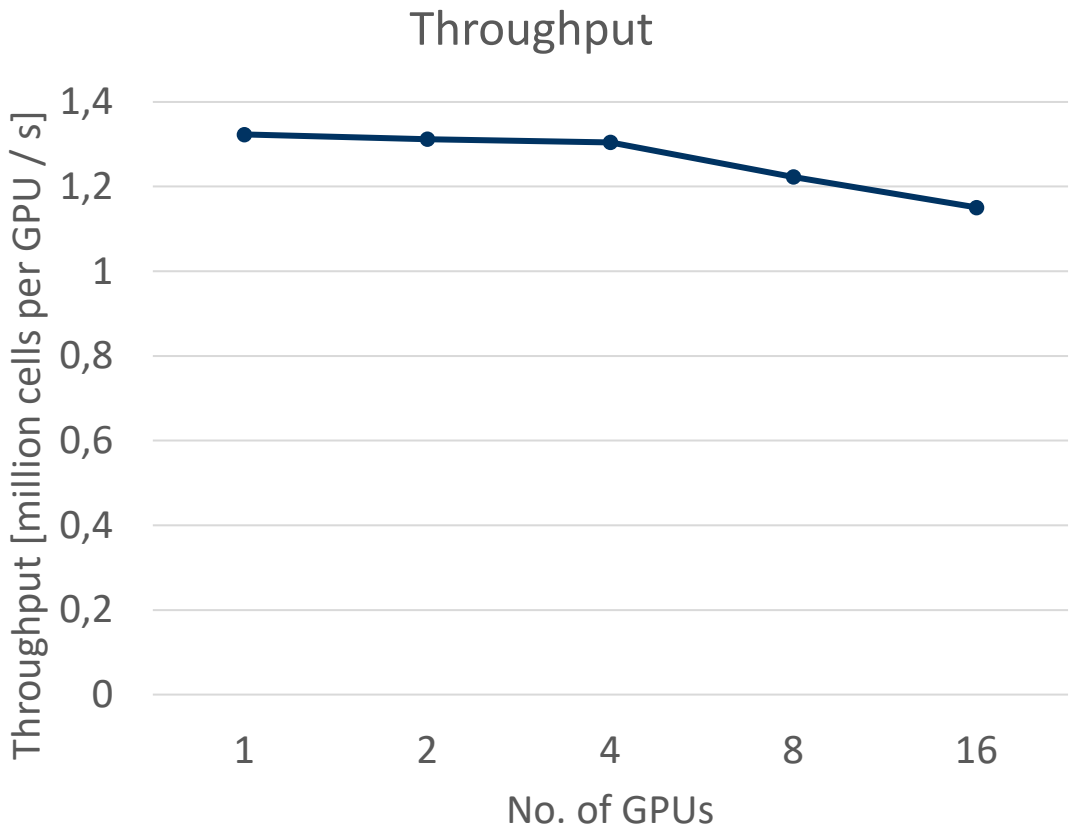
```
matrix.setLow(-5);
matrix.setHigh(5);
matrix[-3][2] = ...;
```

Main code

```
queue.submit([=](celerity::handler& cgh) {

  auto srcAcc = src.get_access<mode::read>(cgh,
    celerity::access::neighborhood<3>(2,2,2));
  auto dstAcc = dst.nom.get_access<mode::write>(cgh,
    celerity::access::one_to_one<3>());

  Problem problem = {...};

  cgh.parallel_for<class Compute>(range,
    sycl::id<3>{3,3,3}, [=](sycl::id<3> item) {
    int z = item.get(0);
    int y = item.get(1);
    int x = item.get(2);

    numValsType numVals[DirMax], numValsX[DirMax];
    numValsType numValsY[DirMax], numValsZ[DirMax];

    compute(srcAcc, numVals, problem, x, y, z);

    // x direction
    compute(srcAcc, numValsX, problem, x + 1, y, z);
    getChanges(dstAcc, problem, numVals, numValsX,
      x, y, z, DirX);

    // y direction
    compute(srcAcc, numValsY, problem, x, y + 1, z);
    getChanges(dstAcc, problem, numVals, numValsY,
      x, y, z, DirY);

    // z direction
    compute(srcAcc, numValsZ, problem, x, y, z + 1);
    getChanges(dstAcc, problem, numVals, numValsZ,
      x, y, z, DirZ);
  });
});
```

Scalability (IFI cluster, AMD EPYC, RTX 2070, 10 GbE)

No. of GPUs	Wall Time [s]	Parallel Efficiency
1	$24.77 \pm 1.55 \times 10^{-3}$	1.00
2	$12.49 \pm 6.91 \times 10^{-4}$	0.99
4	$6.28 \pm 8.12 \times 10^{-4}$	0.99
8	$3.35 \pm 1.08 \times 10^{-3}$	0.93
16	$1.78 \pm 3.77 \times 10^{-4}$	0.87

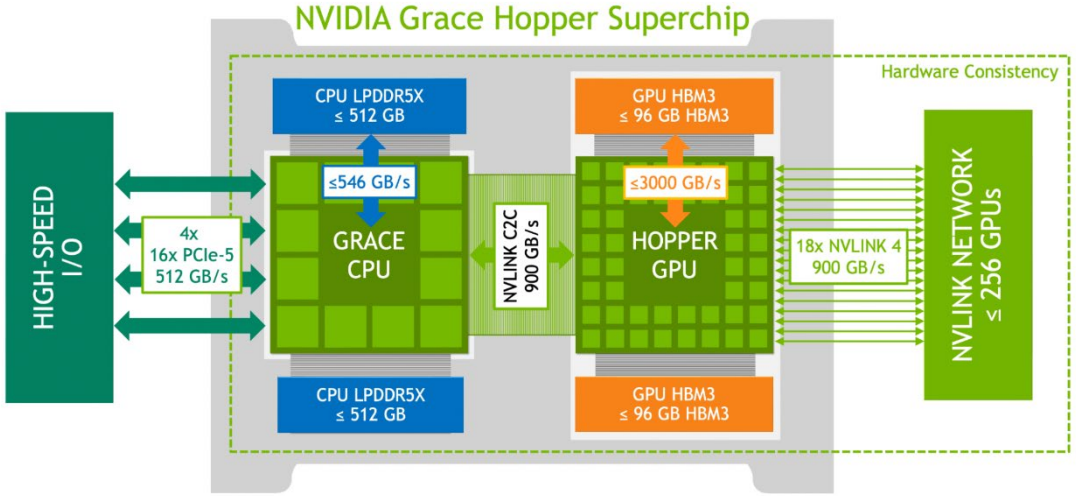


General good practice with accelerators

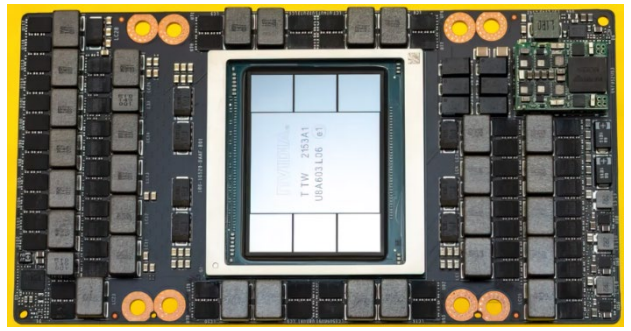
- In order to obtain high performance, you need to “live” on the accelerator
  - avoid data transfers between device and host whenever possible
  - PCI Express is fast, but it’s not that fast
    (e.g. Nvidia RTX 4090 memory bandwidth ~1 TB/s, PCIe 4.0 16x bandwidth ~32 GB/s)
    (e.g. Nvidia H100 memory bandwidth ~3 TB/s, PCIe 5.0 16x bandwidth ~64 GB/s)
- Try to implement algorithms with structured, regular, localized patterns
  - similar to SIMD code, just on a much larger scale
  - use contiguous memory access whenever possible
  - GPUs are not designed for irregular codes
  - highly local data access patterns preferred (e.g. stencils) over global data access (e.g. reductions)
- Do not blindly port every application to accelerators
  - at least consider others, e.g. FPGAs
  - often: simply stick to CPUs, they’re not that bad!

Current hardware trends

- stronger integration of CPU and GPU
  - e.g. Nvidia Grace Hopper: CPU + GPU in a single package
  - memory/cache consistency is a hot topic
- GPU/MPI RDMA
  - allow MPI calls to directly access GPU memory without host RAM detour
- HBM
  - stacked high-bandwidth memory



https://developer.nvidia.com/blog/nvidia-grace-hopper-superchip-architecture-in-depth/



https://wccftech.com/nvidia-hopper-h100-gpu-pictured-worlds-first-4nm-hbm3-chip-for-datacenters/

Conclusion

- GPUs are very capable at massive parallelism
  - but for suitable problems only
- SYCL is a modern programming model
  - main competitor to CUDA, vendor-agnostic
  - makes it easier to work with accelerators
  - still requires some effort to get correct applications AND decent performance